CS422 - Programming Language Design

Two Rewrite Logic Based Programming Language Definitional Styles

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

We next show two additional programming language definitional styles, neither of them following the SOS approach to defining languages.

We exemplify these styles using the same simple language that we used to exemplify the SOS definitions.

Both language definitions below use the same syntax and state modules as the previous two, SOS definitions.

Rewriting Functional Style

The rewrite-based functional approach to define a programming language is the *most straightforward* rewrite-based, or equational, style to define a programming language.

Assuming one's familiarity with term rewriting and/or equational reasoning and assuming also that one did not want purposely to obey an SOS definitional style using rewriting (as we did in the previous two lectures), then one would most likely define a (simple) programming language following this functional style (perhaps without even knowing it :-).

However, we will see that this style suffers from the same problems as the big-step SOS, in particular, it is not suitable to define concurrent languages or languages with complex control statements (halt, exceptions, etc.).

The idea is to find a rewrite-based "implementation" of a function

$$\textit{(Syntactic Category)} \times \textit{State} \rightarrow \textit{Result}$$

that takes a term in any syntactic category, such as an arithmetic or a boolean expression or a statement or a program in our case, and a state, and returns the result obtained after evaluating the given term in the given state.

This would also be the style followed by one who wants to *implement rapidly an interpreter* for a (simple) language using a functional language (and not only), where one would use recursive function calls instead of rewriting.

To keep it similar in notation to the previous two, SOS-based definitional styles, we use a configuration-like notation:

```
fmod CONFIGURATION is
  including STATE .
  op <_,_> : AExp State -> Int .
  op <_,_> : BExp State -> Bool .
  op <_,_> : Stmt State -> State .
  op <_> : Pgm -> Int .
endfm
```

A term `< E,S >` is thought of as a "function call" that evaluates `E` in state `S`.

Having this functional approach in mind, we can now trivially define the evaluation of arithmetic expressions as follows:

```
fmod AEXP-RULES is
  including CONFIGURATION .
  var X : Var .  var S : State .
  var I : Int .  var E1 E2 : AExp .

  eq < X,S > = S[X] .
  eq < # I,S > = I .
  eq < E1 + E2,S > = < E1,S > + < E2,S > .
  eq < E1 - E2,S > = < E1,S > - < E2,S > .
  eq < E1 * E2,S > = < E1,S > * < E2,S > .
  eq < E1 / E2,S > = < E1,S > quo < E2,S > .
endfm
```

Next we define the evaluation of boolean expressions:

```
fmod BEXP-RULES is
  including CONFIGURATION .
  var S : State . var A1 A2 : AExp . var B B1 B2 : BExp .
  eq < true,S > = true .
  eq < false,S > = false .
  eq < A1 <= A2,S > = < A1,S > <= < A2,S > .
  eq < A1 >= A2,S > = < A1,S > >= < A2,S > .
  eq < A1 == A2,S > = < A1,S > == < A2,S > .
  eq < B1 and B2,S > = < B1,S > and < B2,S > .
  eq < B1 or B2,S > = < B1,S > or < B2,S > .
  eq < not B,S > = not < B,S > .
endfm
```

The evaluation of statements follows a similar approach:

```
fmod STMT-RULES is
  including CONFIGURATION .
  var S : State .  var X : Var .  var A : AExp .
  var St St1 St2 : Stmt .  var B : BExp .

  eq < skip,S > = S .
  eq < X := A,S > = S[X <- < A,S >] .
  eq < St1 ; St2,S > = < St2, < St1,S > > .
  eq < { St },S > = < St,S > .
  eq < if B then St1 else St2,S >
   = if < B,S > then < St1,S > else < St2,S > fi .
  eq < while B St,S >
   = if < B,S > then < St ; while B St,S > else S fi .
endfm
```

Program evaluation is also straightforward:

```
fmod PGM-RULES is
  including CONFIGURATION .
  var St : Stmt . var A : AExp .
  eq < St ; A > = < A,< St,empty > > .
endfm

fmod FUNCTIONAL-SEMANTICS is
  including AEXP-RULES .
  including BEXP-RULES .
  including STMT-RULES .
  including PGM-RULES .
endfm
```

Advantages and Disadvantages of the Functional Style

The functional style is similar in spirit to big-step SOS, in the sense that the "function" `< E,S >` evaluates to a value v in the functional semantics if and only if $\langle E, S\rangle \Downarrow v$ is derivable in the big-step SOS. However, it reduces the "big-step" to a sequence of "small-steps" of ordinary rewriting.

Advantages:

- Simple
- Fast
- No conditional equations, so it works on any rewrite engine

Disadvantages:

- Like big-step SOS, it is not suitable for defining languages that have complex control statements, such as halt, exceptions, break/continue of loops, etc.
- Since configurations are expected to evaluate as functions, it is not suitable for defining concurrent languages.
- Definitions of languages become slightly more involved if expressions have side effects; that's because in this case states need to be propagated when evaluating expressions:

```
...
  eq < E1 + E2,S > = < E1,S > + < E2,state(E1,S) > .
  eq state(E1 + E2,S) = state(E2,state(E1,S))
...
```

Continuation Style

The next language definitional style, that we call the *continuation-based style*, is the style that we will use in this class for defining programming languages.

It may appear more complex than the others at first sight; however, as seen in the next HW exercise and in the rest of the class, it overcomes the limitations of the other definitional styles. In particular, it will allow us to define arbitrarily complex control-intensive statement, as well as concurrent programming languages, at no additional effort.

We are not going to discuss this style in more depth here, because we are going to do it in the next lectures. For now, it suffices to be aware of the following:

- The program is flattened in a continuation structure of sort `K`;
- The various syntactic categories and intermediate values, as well as lists of these, are wrapped by corresponding operations in the continuation;
- The continuation structure contains at any moment some intermediate variant of the executing program in postorder representation using the continuation constructor `->` ;
- The *state*, or the *configuration* of the "executing" program contains several attributes in a set "soup", including the continuation and the store; later in the class we will add many other attributes to the state;

```
mod VAL is
  including INT .
  sort Val .
  op bool : Bool -> Val .
  op int : Int -> Val .
endm


mod VAL-LIST is
  including VAL .
  sort ValList .
  subsort Val < ValList .
  op _,_ : ValList ValList -> ValList [assoc id: .] .
  op . : -> ValList .
endm
```

```
mod K is
  including SYNTAX .
  including VAL-LIST .
  sort Kitem K .
  subsort Kitem < K .
  op aexp : AExp -> K .
  op bexp : BExp -> K .
  op stmt : Stmt -> K .
  op pgm : Pgm -> K .

  sort KList .
  subsort K < KList .
  op _,_ : KList KList -> KList [assoc id: .] .
  op . : -> KList .

  op kList : KList -> Kitem .
```

```
  op valList : ValList -> Kitem .
  op _->_ : K K -> K [assoc id: nothing gather (e E) ] .
  op nothing : -> K .
endm

mod CONFIGURATION is
  including K .
  including STATE .
  sort Configuration .
  op __ : Configuration Configuration -> Configuration
            [assoc comm id: .] .
  op . : -> Configuration .

  op k : K -> Configuration .
  op store : State -> Configuration .
endm
```

```
mod K-BASIC is
  including CONFIGURATION .
  op kv : KList ValList -> Kitem .
  var Ke : K . var Kel : KList . var K : K .
  var V : Val . var Vl : ValList .

  eq k(kList(Ke,Kel) -> K) = k(Ke -> kv(Kel,.) -> K) .
  eq valList(V) -> kv(Kel,Vl) = kv(Kel,Vl,V) .
  eq k(kv(Ke,Kel,Vl) -> K) = k(Ke -> kv(Kel,Vl) -> K) .
  eq k(kv(.,Vl) -> K) = k(valList(Vl) -> K) .
endm
```

```
mod AEXP-K is
  including CONFIGURATION .
  var I I1 I2 : Int .  var X : Var .  var K : K .
  var Store : State .  var A1 A2 : AExp .

  eq aexp(# I)  = valList(int(I)) .
  rl k(aexp(X) -> K) store(Store)
   => k(valList(int(Store[X])) -> K) store(Store) .

  ops + - * / : -> Kitem .
  eq aexp(A1 + A2) = kList(aexp(A1),aexp(A2)) -> + .
  rl valList(int(I1),int(I2)) -> + => valList(int(I1 + I2)) .

  eq aexp(A1 - A2) = kList(aexp(A1),aexp(A2)) -> - .
  rl valList(int(I1),int(I2)) -> - => valList(int(I1 - I2)) .
```

```
  eq aexp(A1 * A2) = kList(aexp(A1),aexp(A2)) -> * .
  rl valList(int(I1),int(I2)) -> * => valList(int(I1 * I2)) .

  eq aexp(A1 / A2) = kList(aexp(A1),aexp(A2)) -> / .
  crl valList(int(I1),int(I2)) -> /
   => valList(int(I1 quo I2))
   if I2 =/= 0 .
endm

mod BEXP-K is
  including CONFIGURATION .
  var X : Var . var Store : State . var B B1 B2 : BExp .
  var A1 A2 : AExp . var I1 I2 : Int . var T T1 T2 : Bool .

  eq bexp(true)  = valList(bool(true)) .
  eq bexp(false)  = valList(bool(false)) .
```

```
ops <= >= == and or not : -> Kitem .
eq bexp(A1 <= A2) = kList(aexp(A1),aexp(A2)) -> <= .
rl valList(int(I1),int(I2)) -> <=
=> valList(bool(I1 <= I2)) .

eq bexp(A1 >= A2) = kList(aexp(A1),aexp(A2)) -> >= .
rl valList(int(I1),int(I2)) -> >=
=> valList(bool(I1 >= I2)) .

eq bexp(A1 == A2) = kList(aexp(A1),aexp(A2)) -> == .
rl valList(int(I1),int(I2)) -> ==
=> valList(bool(I1 == I2)) .

eq bexp(B1 and B2) = kList(bexp(B1),bexp(B2)) -> and .
rl valList(bool(T1),bool(T2)) -> and
=> valList(bool(T1 and T2)) .
```

```
  eq bexp(B1 or B2) = kList(bexp(B1),bexp(B2)) -> or .
  rl valList(bool(T1),bool(T2)) -> or
  => valList(bool(T1 or T2)) .

  eq bexp(not B) = bexp(B) -> not .
  rl valList(bool(T)) -> not => valList(bool(not T)) .
endm

mod STMT-K is
  including CONFIGURATION .

  var X : Var .  var A : AExp .  var St St1 St2 : Stmt .
  var B : BExp .  var K1 K2 K : K .
  var I : Int . var Store : State .

  eq stmt(skip) = nothing .
```

```
eq stmt(X := A) = aexp(A) -> write(X) .

op write : Var -> Kitem .
rl k(valList(int(I)) -> write(X) -> K) store(Store)
 => k(K) store(Store[X <- I]) .

eq stmt(St1 ; St2) = stmt(St1) -> stmt(St2) .

eq stmt({St}) = stmt(St) .

op if : K K -> Kitem .
eq stmt(if B then St1 else St2)
 = bexp(B) -> if(stmt(St1),stmt(St2)) .
rl valList(bool(true)) -> if(K1,K2) => K1 .
rl valList(bool(false)) -> if(K1,K2) => K2 .
```

```
  op while : K K -> Kitem .
  eq stmt(while B St) = bexp(B) -> while(bexp(B),stmt(St)) .
  rl valList(bool(true)) -> while(K1,K2)
  => K2 -> K1 -> while(K1,K2) .
  rl valList(bool(false)) -> while(K1,K2)
  => nothing .
endm

mod PGM-K is
  including K .
  var St : Stmt . var A : AExp .
  eq pgm(St ; A) = stmt(St) -> aexp(A) .
endm
```

```
mod EVAL is
  including K-BASIC .
  including AEXP-K .
  including BEXP-K .
  including STMT-K .
  including PGM-K .

  op <_> : Pgm -> Int .
  op result : Configuration -> Int .
  var P : Pgm . var I : Int . var Store : State .
  eq < P >
   = result(k(pgm(P)) store(empty)) .
  eq result(k(valList(int(I))) store(Store) )
   = I .
endm
```

Homework Exercise

There is only one HW exercise in this block of lectures, which is described below. There will be no HW exercise in the next lecture. As expected, this HW exercise is more complex than the others.

Homework Exercise 1 *You are required to add two language constructs to our simple language:*

- `inc : Var -> AExp`, *which evaluates to the value of the variable and then increments the value of that variable; this is like ++ in Java or C++; and*
- `halt : AExp -> Stmt`.

Define the semantics of the two language constructs in each of the four language definitions discussed in the class. You are supposed to modify the provided Maude codes. Return four printouts, one for each definitional style, including both additional definitions.